Parallel Computing 117 (2023) 103042

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

# Finding inputs that trigger floating-point exceptions in heterogeneous computing via Bayesian optimization

Ignacio Laguna [a,*], Anh Tran [b], Ganesh Gopalakrishnan [b]

[a] Lawrence Livermore National Laboratory, CA, United States of America
[b] University of Utah, UT, United States of America

ARTICLE INFO

*Keywords:*
Floating-point exceptions
GPU computing
Bayesian optimization

ABSTRACT

Testing code for floating-point exceptions is crucial as exceptions can quickly propagate and produce unreliable numerical answers. The state-of-the-art to test for floating-point exceptions in heterogeneous systems is quite limited and solutions require the application's source code, which precludes their use in accelerated libraries where the source is not publicly available. We present an approach to find inputs that trigger floating-point exceptions in black-box CPU or GPU functions, i.e., functions where the source code and information about input bounds are unavailable. Our approach is the first to use Bayesian optimization (BO) to identify such inputs and uses novel strategies to overcome the challenges that arise in applying BO to this problem. We implement our approach in the XSCOPE framework and demonstrate it on 58 functions from the CUDA Math Library and 81 functions from the Intel Math Library. XSCOPE is able to identify inputs that trigger exceptions in about 73% of the tested functions.

## 1. Introduction

Testing numerical applications is a significant challenge for most scientific computing developers. Most applications use floating-point arithmetic, compilers optimizations can drastically affect floating-point results, and developers must ensure that round-off error does not negatively impact numerical results. While controlling rounding error is essential, floating-point exceptions, such as division by zero, infinity or Not a Number (NaN), must also be detected and either eliminated or reduced below acceptable thresholds.

In heterogeneous computing programs (composed of GPU and CPU code), tools to detect or predict floating-point exceptions are scarce and today developers lack practical solutions to the problem. Recent work can detect such exceptions in CPU programs by reading hardware-level registers that are set when the exception occurs [1]. However, such methods are hardware-dependent, and are limited in their ability to trace back the exact line of code that caused the exception and/or the input values that triggered it. In NVIDIA GPUs—the most used GPUs in HPC systems—the situation is even worse. NVIDIA GPUs do not provide such register flags and CUDA provides no mechanism to detect such exceptions.[1] Compiler-based solutions, such as [2], require sources and can detect exceptions at runtime, but only for *given inputs*. Ideally, developers would want to know the **inputs** that induce exceptions so those inputs can be controlled and explored systematically during testing.

There is prior work on identifying inputs that induce floating-point exceptions in CPU programs [3]. While these methods could (in principle) be implemented on GPU kernels, their drawback is that they use SMT solvers and symbolic execution, which requires analyzing the source code. Unfortunately, practical GPU codes running on NVIDIA GPUs involve using proprietary accelerated libraries, such as cuBLAS, cuFFT, cuSOLVER, CUDA Math Lib, cuTENSOR, cuSPARSE, and cuDNN [4], for which the source code is not publicly available. Even popular machine learning frameworks, such as PyTorch [5], make heavy use of cuBLAS and cuDNN. As a result, methods that require the source code are limited to test accelerated libraries or code that use them.

**Our Contributions.** This paper presents XSCOPE,[2] a framework to find inputs that trigger floating-point exceptions in heterogeneous program functions, i.e., in a function $f(x)$, where the function user has limited knowledge of how the function operates. More specifically, the source of $f(x)$ is not available—the function is a black box from the user's perspective—and the user does not know a priori the input bounds that the function expects, i.e., inputs can be any

* Corresponding author.
*E-mail address:* lagunaperalt1@llnl.gov (I. Laguna).
*URL:* https://lagunaresearch.org/ (I. Laguna).

[1] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[2] Floating-point eXceptions-scope

https://doi.org/10.1016/j.parco.2023.103042
Received 18 January 2023; Received in revised form 5 May 2023; Accepted 27 July 2023
Available online 2 August 2023

normal floating-point number. Our method relies on using Bayesian optimization (BO) to explore $f$ in a guided manner with the objective of pinpointing extreme cases in $f$. These extreme cases make $f$ return the result of exceptions to the user, i.e., infinity (positive and negative), underflows (i.e., subnormal numbers), or NaN (not a number). Finally, the user is provided the inputs of $f$ that triggered such exceptions. Developers can also use XSCOPE to test functions where the code is available and/or input bounds are known, which only improves XSCOPE's chances of success. *To the best of our knowledge, we are the first to apply BO to identify inputs that trigger floating-point exceptions.*

While BO is well suited to optimize black-box functions [6] and fits our problem well, we encounter two significant challenges when directly applying BO to our problem. The first is that BO naturally maximizes (or minimizes, depending on how it is implemented) a given a function, and can find inputs that return infinity (positivity or negative) but cannot find inputs that return underflows—the region of floating-point underflows is not necessarily in the extremes of an arbitrary function. We propose a method called *function twisting* that transforms $f$ into another function $f'$ so that maximizations/minimizations are guided into the underflows regions. Function twisting prevents BO from getting stuck in zero or a smaller number when BO attempts to search for underflows.

The second problem is that, since we assume the space of possible inputs is unbounded, there is no guarantee that BO identifies the global maxima or minima [7,8]. We tackle this challenge using two divide-and-conquer approaches, called *input range splitting* and *exponent sampling*. The first allows BO to try different input bounds on different function parameters in an iterative fashion, which improves its chances of success in triggering exceptions. The second samples exponents of floating-point numbers, as opposed to sampling the numbers per se—since the range of possible exponents is smaller, this can speedup the search. In summary:

- We present the first approach to identify inputs that trigger floating-point exceptions in GPU functions for which the source code is not available and information about input bounds is limited (or unavailable). Our approach is the first to apply BO to search for such inputs. We present two strategies to address the main challenges in using BO to this problem: guiding BO to explore the floating-point underflows region and helping BO identify global minima/maxima for unbounded inputs.
- We implement our methods in the XSCOPE framework and test it in Intel CPUs and NVIDIA GPUs. XSCOPE takes as input the signature of the function $f(x)$ to test, generates a wrapper CUDA code or C code $w(x)$ that calls $f(x)$ and that can be passed as target to the BO optimizer (a Python module). We design and present an algorithm that implements the function transformations and considers different approaches to sample numbers. When XSCOPE finishes, it provides a report to the user about the category of the exceptions found and the inputs that triggered them.
- We demonstrate the usefulness of XSCOPE in 58 double-precision functions from the CUDA Math Library[3] and 81 functions from the Intel Math Library for which the source is unavailable and on functions from ten HPC programs for which the source is available. XSCOPE identifies inputs that trigger exceptions in 72% of the tested functions from the CUDA Math Library and 75% of the tested functions from the Intel Math Library. We also present cases where XSCOPE's findings augment the specification provided in the library documentation—for example, XSCOPE indicates that infinity is generated in a function $f$ with a specific input, but the documentation of $f$ does not specify the behavior for such input.
- We present a comparison of the inputs found in functions that are provided in both CPU and GPU implementations of the Math Library. XSCOPE is able to identify a few functions that exhibit inconsistent behavior between the CPU and GPU implementations.

[3] https://developer.nvidia.com/cuda-math-library

**Table 1**
Floating-point exceptions as defined by the IEEE 764 standard.

| Event | Description |
|---|---|
| `Inexact` | Result is produced after rounding |
| `Underflow` | Result could not be represented as normal |
| `Overflow` | Result did not fit and it is an infinity |
| `DivideByZero` | Divide-by-zero operation |
| `Invalid` | Operation operand is not a number (NaN) |

**Comparison to Previous Work.** A previous version of this idea was published in [9]. The current methodology has been significantly improved and new results have been obtained, which reveal insights not published before. In particular:

- The new XSCOPE approach produces more accurate results in finding inputs that trigger underflows (subnormal numbers) and infinity. The earlier algorithm used in [9] reported inputs caused negative infinity in functions that could never return such cases, for example, `exp(x)`. The algorithm in this paper avoids such cases for improved accuracy.
- The method in [9] reported duplicated inputs for the same exception. As a consequence, the results presented here are much more accurate than those presented in [9]. The current version only reports unique inputs that cause exceptions.
- The previous version was designed only for GPU programs. We have extended the framework to generate tests for CPU and GPU code, which allows us to test functions in both devices.
- We present results from the Intel Math Library, which runs in the CPU—the previous version in [9] only considered the CUDA Math Library. We compare the results from both libraries and present interesting results not presented in [9].

## 2. Background and overview

In this section, we provide background on floating-point exceptions in heterogeneous systems, and give a high-level description of BO, which will be useful to understand our contributions. We refer the reader to [6,10] for a more detailed description of BO. Then, we describe the challenges of applying BO to triggering exceptions and, finally, an overview of XSCOPE.

### 2.1. IEEE 754 exceptions

We recap the IEEE 754 Standard for floating-point arithmetic and exceptions—all fundamental to explaining how XSCOPE approaches these. A floating-point number has the form

$$x = \pm s \times \beta^e \tag{1}$$

where the sign, the significant $s$, the exponent $e$ can be stored in memory or a register. We assume $\beta = 2$, since this is the most used format for representing floating-point numbers.

The IEEE 754 Standard defines five classes of *exceptions*, that can result from arithmetic operations. Table 1 shows these five events. When one of these events occurs, the floating-point unit can set a status register specifying which event occurred. Existing frameworks for CPU analysis, such as [1] read these registers to detect the occurrence of such events. Additionally, with the help of the compiler and system routines, they raise a floating-point exception signal (e.g., `SIGFPE`) when these events occur in the CPU. Unlike CPUs, NVIDIA GPUs have no mechanism to detect floating-point exceptions, set a status register or raise a signal when an exception occurs.

**Exceptional Quantities.** Except for the `Inexact` exception in Table 1, the rest of the events will result in either a `NaN` (not a number), `INF` (infinity, positive or negative), or a subnormal number (i.e., a number smaller than a normal floating-point number but that is not

zero). More specifically, `Overflow` and `DivideByZero` result in `INF`, and `Invalid` result in `NaN`. `Underflow` can result in zero or a subnormal number—in our case, we are interested in underflows that result in subnormal numbers. The `Inexact` event results in a rounding operation; however, this occurs frequently in numerical programs and it is usually of no interest to programmers. In summary, our goal is to identify inputs that produce any of these cases: `NaN`, `INF+`, `INF-`, or subnormal number quantities (positive or negative).

**Subnormal Numbers.** Note that while subnormal numbers can represent specific real number quantities, they are often dangerous for several reasons. First, they indicate that computational results are becoming too small to be represented in the current precision. Second, if they propagate to the denominator of a division, the result can produce `INF`. For example, $\frac{1}{x}$, where $x$ =1e−309 (a FP64 subnormal number), produces `INF`[4]. Third, when subnormal numbers are combined with compiler optimizations, they can cause reproducibility issues [11]. Therefore, it is crucial to mitigate them and understanding when they occur as well as `NaN` and `INF`.

### 2.2. Bayesian optimization

Bayesian optimization (BO) is a class of machine-learning-based optimization methods focused on solving the problem

$$\max_{x \in A} f(x), \tag{2}$$

where $A$ is the feasible input set, which is typically a hyper-rectangle $\{x \in \mathbb{R}^d : a_i \leq x_i \leq b_i\}$, i.e., it is a bounded optimization method. BO is useful in situations when (a) $f$ is a black box for which no closed form is known (nor its gradients), (b) $f$ is expensive to evaluate, and (c) evaluations of $y = f(x)$ may be noisy. The function $f$ is also known as the *target function*. In the case of the problem we want to tackle, we define target functions as program functions executed in the CPU or GPU, for which the code may or may not be available, where the inputs and outputs are floating-point numbers.

**BO Algorithm in a Nutshell.** BO attempts to find the global maxima of $f$ through a method known as *surrogate optimization*—a surrogate function is an approximation of the target function (which is unknown). Based on the surrogate function, BO identifies which points are a promising maxima, and decides to sample more from these promising regions and update the surrogate function accordingly. In each iteration, it continues to look at the current surrogate function, learn more about areas of interest by sampling, and update the function. After a certain number of iterations, BO is expected to arrive to the global maxima of $f$[5].

The surrogate function is represented by a Gaussian processes (GP), which can be thought of as a probabilistic function that returns several functions with probabilities attached to them. The GP models the target function with the observed information, i.e., is Bayesian in nature. The surrogate function is updated with an *acquisition function*, which is responsible for suggesting new points to test. The acquisition has an exploration and exploitation trade-off: *exploitation* seeks to sample where the surrogate model predicts a good objective; *exploration* seeks to sample in locations with high uncertainty.

### 2.3. Challenges of applying BO

Let us consider a CPU/GPU function with the following signature

```
__device__
double compute(double x, ...);
```

[4] This behavior may depend on the platform, compiler, and optimizations applied to the code.

[5] Unless the function's shape is very unconventional (e.g., it has large and sudden swings), on which case it may find a local maxima.

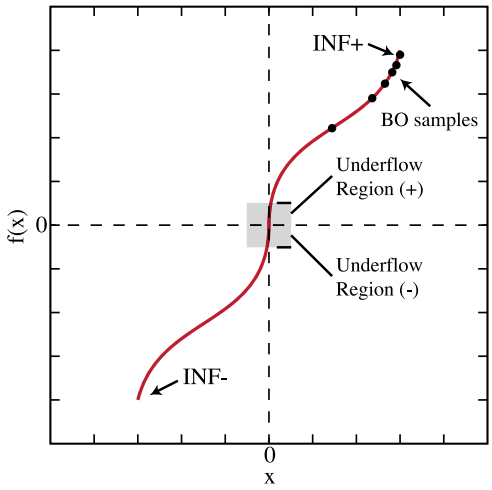


**Fig. 1.** Sample function $f(x)$. When BO searches for the $f$ maxima it can arrive to `INF+`; alternatively when it searches for the minima, `INF-` can be found. However, the underflow regions (positive and negatives) are missed.

The function has a scalar input $x$, could have other non-floating-point input parameters, and returns a scalar output. The input and output is of FP64 (double precision) type. We map `compute(double x, ...)` to $f(x)$ and use BO to identify extreme cases with the hope that such cases correspond to floating-point exceptional quantities. At each iteration point, BO provides both the input and the maxima of $f(x)$ observed so far. Note that we may or may not have the source code of the function. Let us suppose that Fig. 1 shows the output of the function with respect to the input.

**Major Difficulties and Solutions.** We now encounter two major difficulties on identifying inputs that trigger exceptional floating-point quantities. The first problem we encounter is that BO cannot identify inputs that trigger subnormal numbers (which are the result of underflows) in the function. The reason is that the regions of subnormal numbers are in the middle of the returned values domain. In other words, if the function is maximized, we may reach the `INF+` point and if the function is minimized we may reach the `INF-` point; however, neither the maximization nor minimization loop in BO would stop at any point in the subnormal regions (since there will always be a larger or smaller point, respectively).

We tackle the above problem using a method we call *function twisting*. The idea is to transform the function so that underflow regions are seen as minima or maxima to BO by flipping a quadrant of the function up or down (as if a portion the plot in Fig. 1 is twisted over the $x$ axis but not the entire plot—more on Section 3.3). We need to consider the zero case carefully (+0.0 and −0.0). For example, if we are minimizing, BO could believe that 0.0 is the smallest value; however, subnormal numbers are very close to zero, but they are not zero (e.g., 1e−309 for double precision) and we would like BO to return a subnormal number as the minima, not zero.

The second problem we encounter is that, since we assume that users do not have prior knowledge of the input bounds for $x$, there is no guarantee BO finds the global optimum. This is a known problem—application of the BO framework when the search region is unknown remains an open challenge in the community [7,8]. We tackle this challenge using a divide-and-conquer approach (inspired by interval analysis [12]) where instead of asking BO to find extremes in an unbounded range—where an input $x$ can be any floating-point number between the minimum and maximum normal number—we ask BO to operate over combinations of intervals in that range. It turns out that this helps BO identify more interesting inputs at the cost of more iterations (see Section 3.4).

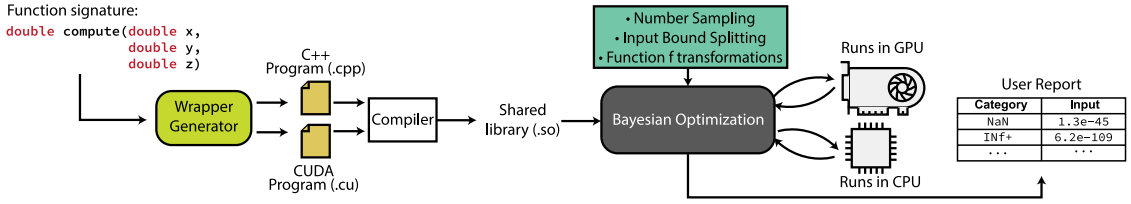


**Fig. 2.** Overview of XSCOPE's workflow.

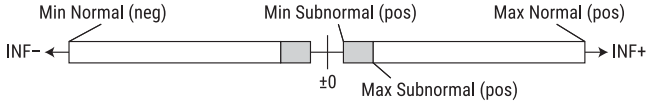


**Fig. 3.** Location of max and min of normal and subnormal numbers.

### 2.4. Overview of XSCOPE

Fig. 2 shows an overview of XSCOPE's workflow. We start by the user providing the signature of the function to test, which includes the return type and the input parameters. After this, XSCOPE generates a C++ program (.cpp) or CUDA program (.cu) that contains: (a) a kernel wrapper that calls the function to test and (b) a C-linkage function that calls the kernel wrapper. Next, everything is compiled and linked into a shared library by the compiler (for CUDA we use nvcc). Alternatively, the user can provide the shared library directly that already contains the C-linkage function. Later the shared library is loaded in Python and a pointer of the C-linkage function is passed to the BO engine (which is written in Python).

The BO framework works by taking as input several methods to sample floating-point numbers (see Section 3.4) and considering different input bounds. It then search for inputs that generate exceptions and, when it finds them, it provides a report to the user about the category of the exceptions and the inputs that triggered them.

**XSCOPE's Practical Utility.** XSCOPE has a wide range of practical applications. First, given specific exception-triggering inputs, developers can add assertions in the code to check for such inputs before they propagate to the functions that trigger the exception—currently, there are few tools (if any) to help developers create such assertions. Second, the exception-triggering inputs can be used in testing campaigns to strengthen the code and understand how program inputs affect internal functions or code fragments—today, GPU developers are in need for such tools to help test GPU code. Third, such "dangerous" inputs can be documented better so that users of a library API avoid them. Fourth, when exceptional quantities (e.g., `NaN`, `INF`) are combined with high compiler optimizations (e.g., `-fast-math`), they can introduce numerical reproducibility issues [11]; XSCOPE can help in isolating inputs that lead to this behavior.

## 3. Approach

We present the technical aspects of our approach.

### 3.1. Floating-point number ranges

Before describing our method, it is useful to review the floating-point input ranges allowed by the IEEE 754 standard for floating-point arithmetic. We focus on double precision since it is the precision most used in scientific computing applications; however, our methods can be applied in other precisions (e.g., FP32 or FP16). Our approach needs these ranges—and the minimum and maximum values—to perform the unbounded search in the inputs of $f$. Table 2 shows the minimum and maximum values for normal and subnormal numbers (positive and negative). Fig. 3 illustrates their location. Note that a computation that produces a value greater than $N^+_{max}$ (i.e., the max normal positive) will result in `INF+`.

**Table 2**
Minimum and maximum floating-point values for double precision (FP64).

| Number | Symbol | Value |
|---|---|---|
| Min Subnormal (positive) | $S^+_{min}$ | $\approx$ 4.941e−324 |
| Max Subnormal (positive) | $S^+_{max}$ | $\approx$ 2.225e−308 |
| Min Normal (positive) | $N^+_{min}$ | $\approx$ 2.225e−308 |
| Max Normal (positive) | $N^+_{max}$ | $\approx$ 1.798e308 |
| Max Subnormal (negative) | $S^-_{max}$ | $-S^+_{min}$ |
| Min Subnormal (negative) | $S^-_{min}$ | $-S^+_{max}$ |
| Max Normal (negative) | $N^-_{max}$ | $-N^+_{min}$ |
| Min Normal (negative) | $N^-_{min}$ | $-N^+_{max}$ |

### 3.2. Function optimization

By default, our BO implementation *maximizes* $f$. To minimize $f$, we maximize $y = -f(x)$. We assume the target function $f$ has one or more floating-point input parameters, and that it returns a scalar double precision value. If the function returns multiple values $\{r_1, r_2, \ldots\}$ (e.g., it writes results to an array), we assume the user can transform the returned values to a scalar, e.g., by applying $\max(r_1, r_2, \ldots)$, or another transformation function.

**Multiple Output Function Example.** Consider the Jacobi Singular Value Decomposition function from Eigen C++ template library for linear algebra [13], which solves the following system of equations: $Ax = b$, where A is the input matrix of shape $n \times m$, b is the $m-dimension$ input vector, and $x$ is the $n-dimension$ output vector.

To test this function, we can wrap it as follow:

```
__device__
double JacobiSVD(MatrixXd A, VectorXd b) {
  JacobiSVD svd(A);
  VectorXd x = svd.solve(b);
  double scalar_x = x.maxCoeff(NaNPropagation=
     PropagateNaN);
  return scalar_x;
}
```

Here we transform the output to scalar floating-point value by applying the `maxCoeff()` function on $x$ with the `NaNPropagation` flag set to return `NaN` upon encounter. Now we can search for inputs that trigger exceptions in the function using XSCOPE as usual. Note that some functions, such as `min` and `max` from the standard C library, have inconsistent behaviors when dealing with `NaN` or `INF`. Thus, a better strategy would be to search through the outputs for exception values first before applying any transformation.

### 3.3. Function twisting

Let us suppose we want to search for inputs that produce subnormal numbers (the result of underflows) in the positive underflow region. Let us assume $f$ is a normalized sigmoid function. The following listing shows the implementation of $f$ and Fig. 4 plots it for the parameter $k = 0.9$:

```
__device__
double sigmoid(double x, double k) {
  double d = x - (k*x);
```

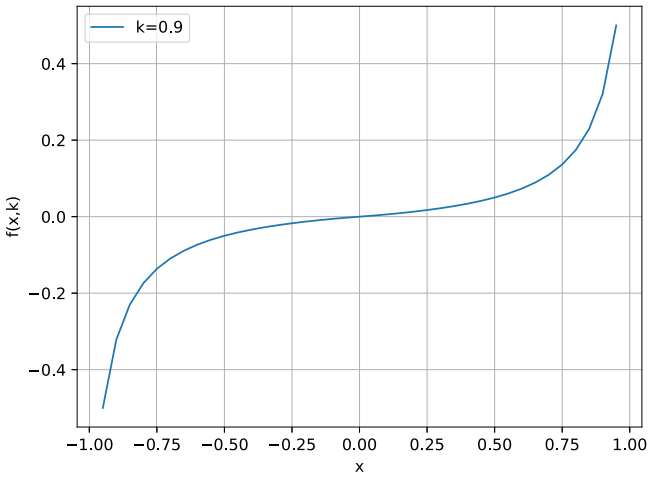


**Fig. 4.** Sample of a normalized sigmoid function.

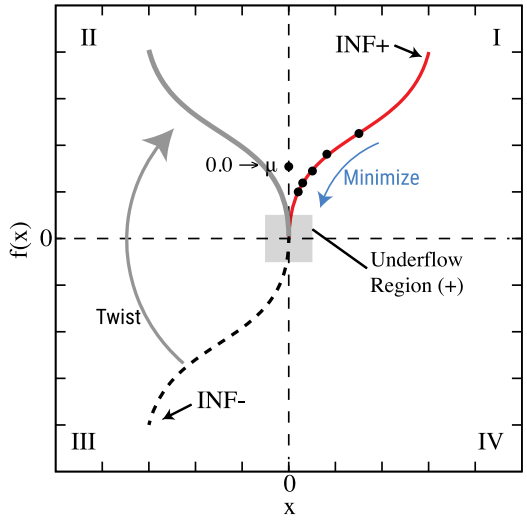


**Fig. 5.** Illustration of the function twisting idea.

```
4   double n = k - 2.0*k*abs(x) + 1.0;
5   return d/n;
6 }
```

Since we are searching for small returned values, we ask BO to minimize $f$. BO might begin trying random points, e.g., $x = 0.75$ and get $f(x) = 0.1363$. It may keep trying lower values following the suggestions of the acquisition functions. For example, it could try $x = 4\text{e}{-308}$, which is a normal number, and get $f(x) = 2.105\text{e}{-309}$. Note that $2.105\text{e}{-309}$ is a subnormal number since it is smaller than $S^+_{max}$. However, there is no guarantee that BO stops at this point since there are even smaller values for $f$ and BO is minimizing the function; for example, $f(0.0) = 0.0$ and $f(-0.1) = -0.00581$. Note also that there is no guarantee that BO even finds $f(x) = 2.105\text{e}{-309}$ since it may jump quickly to smaller values of $f$.

To address this issue, we propose an approach that we call *function twisting*, which allows BO to move towards the positive underflow region and stay in that region until the smallest subnormal number is found, at which point BO has identified the function minima. The idea is to twist the quadrant III of the function to quadrant II so the underflow region is exposed to BO as the smallest region and it stops in that region when minimizing. Fig. 5 illustrates the idea.

In other words, we transform $f$ into a new function $f'$ and ask BO to minimize $f'$. More formally $f'$ is defined as:

$$f'(x) = \begin{cases} \mu & \text{if } f(x) = 0.0 \text{ or } f(x) = -0.0 \\ f(x) & \text{if } f(x) \geq S^+_{min} \\ -f(x) & \text{if } f(x) < S^+_{min}. \end{cases} \tag{3}$$

Note that zero is a special case. Since zero is smaller than $S^+_{min}$, BO could think zero is the smallest point and would not stop at any subnormal number. We assigned zero the parameter $\mu$ with the condition that it must be greater than the largest subnormal, i.e., $S^+_{max}$. In practice, we have found that $\mu = 1.0$ works well.

*What about negative subnormals?* For the negative underflow region we want BO to maximize $f$ and use a similar approach where the quadrant I is flipped down to quadrant IV. More formally:

$$f'(x) = \begin{cases} -\mu & \text{if } f(x) = 0.0 \text{ or } f(x) = -0.0 \\ f(x) & \text{if } f(x) \leq S^-_{max} \\ -f(x) & \text{if } f(x) > S^-_{max}. \end{cases} \tag{4}$$

Note that function splitting is not simply *flipping* the function on the $x$ or $y$ axis since we only flip one quadrant (flipping moves both quadrants) and zero must be handled differently. Also note that for finding inputs that trigger `INF+` and `INF-`, function flipping is not required.

### 3.3.1. Reporting correct exceptions sign

When $f$ returns an exception, we need to consider the sign of the exception carefully. If the exception came from the part of the function that was negated (i.e., $-f(x)$), the sign of the exception must be negated to reflect the correct original sign. We use $-f(x)$ in two situations: (1) when we ask BO to minimize (by default our BO maximizes), and (2) when a part of the function is twisted. This was not considered in [9]. As a result, if $\epsilon$ is an exception returned from $-f(x)$, we correctly record $-\epsilon$.

### 3.4. Input range splitting

Since we assume that the user has limited (or null) a priori knowledge of the input bounds, in the general case, input values can be any normal floating-point number or zero—we discard exceptional cases, such as `INF` or `NaN`, as input since they are usually not used to compute anything. This, however, is a very large input range, which challenges BO and there is no guarantee it can arrive to a global maxima/minima [7,8]. To address this challenge, we use various divide-and-conquer methods to split the input bounds and sample numbers:

- **Whole-range approach**: inputs can be any floating-point number $g \in \mathbb{F}$, where $\mathbb{F}$ is the domain of acceptable input floating-point numbers (i.e., normal numbers or zero).
- **Two-range approach**: we split the whole input range into two ranges: $[N^-_{min}, -0.0]$ and $[+0.0, N^+_{max}]$. Inputs are sampled from any of these two ranges.
- **Many-range approach**: we split the whole input range into many ranges, according to the following:

$$\begin{aligned} F = \{&N^-_{min}, -1\text{e}{+100}, -1\text{e}{+10}, -1\text{e}{+1}, -1\text{e}0, -1\text{e}{-1}, \\ &-1\text{e}{-10}, -1\text{e}{-100}, -1\text{e}{-307}, 0.0, 1\text{e}{-307}, 1\text{e}{-100}, 1\text{e}{-10}, \\ &1\text{e}{-1}, 1\text{e}0, 1\text{e}{+1}, 1\text{e}{+10}, 1\text{e}{+100}, 1\text{e}{+307}, N^+_{max}\} \end{aligned} \tag{5}$$

$$B_{fp} = [g_a, g_b] \mid g_a \in F, g_b \in F, g_a < g_b \tag{6}$$

$$\texttt{sample\_fp}() : \emptyset \rightarrow \mathbb{F} = \{g \in B_{fp}\} \tag{7}$$

In the below ranges, $F$ corresponds to cutting points of $\mathbb{F}$, $B_{fp}$ is the bounds of floating-point input ranges, and `sample_fp()` is the sampling

function.[6] We call this approach the *fp* approach as we sample pure floating-point numbers (hence sets are denoted with the *fp* subindex).

**Sampling Exponents.** The above approach (*fp*) for sampling is computationally expensive, particularly on the *many-range* mode of operation since there are many combinations of ranges to explore. Range combinations increase exponentially with the number of input parameters. We study a different approach, called *exp*, which samples the exponents of floating-points, as opposed to sampling numbers from $\mathbb{F}$. The idea is that the ranges of exponents is approximately $[-307, +307]$, which is significantly smaller than the range of $\mathbb{F}$.

Here, we also consider the whole-range, two-range, and many-range modes of operation. For the whole-range approach, we simply sample exponents in the range of $[-307, +307]$ and create a floating-point number with mantissa $s = 1.0$. For the two-range mode, we split it in two: $[-307, 0]$ and $[0, +307]$. The many-range mode splits the original range in many as follows:

$$E = \{-307, -100, -10, -1, 0, +1, +10, +100, +307\} \tag{8}$$

$$B_{exp} = [e_a, e_b] \mid e_a \in E, e_b \in E, e_a < e_b \tag{9}$$

$$\texttt{sample\_exp}() : \varnothing \rightarrow \mathbb{F} = \{\pm 1.0 \times \beta^e \mid e \in B_{exp}\} \tag{10}$$

Here, $E$ corresponds to cutting points of the original range $[-307, +307]$, $B_{exp}$ is the bounds (or range) of allowed exponents, and `sample_exp()` is the sampling function. Note that this approach will trade off success in identifying exceptional values for speed—as Section 4 shows, it requires running the target function fewer times compared to the *fp* method.

### 3.5. Algorithm

We present the logic for input searching in Algorithm 1. The algorithm requires as input a function pointer of $f$ (so it can be invoked in the BO loop), the number of parameters $p$ of $f$, and the approaches for input splitting and number sampling (*fp* or *exp*). The algorithm loops on the different goals it tries to achieve, i.e., trying to find `INF+`, `INF-`, and positive and negative underflows (lines 5–7). For each goal, it sets $f$ to the corresponding target function depending on whether the goal is achieved by maximizing/minimizing $f$ and using function twisting (lines 8–11).

We then get combinations of bounds, assign them to $B$, depending of the methods we are using (as explained earlier in this section), and iterate on them. $B$ is a list of different input ranges. These bounds are used to set the BO optimizer (line 12). Next, the algorithm enters the BO main loop, on which a point is suggested by the acquisition function, it is evaluated in $f$, and the Gaussian process (and BO algorithm) is updated with the result $v = f(next\_point)$.

At each iteration in the BO loop, we call the function IS_EXCEPTIONAL_NUMBER() to check if the current value is the result of an exception, which could include `INF`, subnormal numbers or `NaN`. If that is the case, we inform the user of the input and result obtained, and break the current loop iteration as we have arrived to the goal.[7]

The algorithmic complexity (worst case) of Algorithm 1 is $O(C \cdot k \cdot r^p)$, where $C$ is the cost of updating the Gaussian process and BO framework at each iteration—this cost is implementation dependent; $k$ is the maximum number of samples explored by BO (denoted as *max_iterations*) in the algorithm; $r$ is the number of inputs ranges (or bounds) explored. $r$ is 1 for the *whole-range* range approach but it is larger for the approaches that split the input in several chunks (e.g., *two-range* and *many-range*).

[6] This choice of $F$ is meant to reflect the clustering of floating-point numbers around 0.

[7] An added reason for breaking out is that adding the input and an exceptional value output on line 18 can result in an ill-defined Gaussian process update.

**Algorithm 1** FindProblematicInputs

```
Require: f : Target function pointer
 1: p : Number of input parameters of f
 2: i : Input splitting method (whole, two, many)
 3: n : Number sampling method (fp, exp)
 4: procedure FIND_INPUTS(f, p, i, n)
 5:     Goals ← {find_inf+, find_inf-,
 6:     find_under+, find_under-}
 7:     for g ∈ Goals do
 8:         if g == find_inf+ then f ← f                    ▷ do nothing
 9:         else if g == find_inf- then f ← −f
10:         else if g == find_under+ then f ← −f′
11:         else if g == find_under- then f ← f′
12:         B ← GET_BOUNDS(i, n, p)                        ▷ create bounds set
13:         for b ∈ B do
14:             SET_BO_BOUNDS(b)
15:             while n < max_iterations do
16:                 next_point ← SUGGEST_NEXT_POINT()
17:                 v ← f(next_point)
18:                 UPDATE_GAUSSIAN_PROCESS(v)
19:                 if IS_EXCEPTIONAL_NUMBER(v) then
20:                     REPORT_USER(v)
21:                     break
22:                 end if
23:             end while
24:         end for
25:     end for
26: end procedure
```

**Unique Input Tracing.** Our method maintains a global table where keys are functions $f$ (i.e., the function name tested) and the input values found to generate exceptions. In some situations, XSCOPE can find duplicated inputs that generate a known exception. Our current implementation maintains a set of unique exception-triggering inputs for each function (previous implementations did not consider this [9]).

### 3.6. Implementation details

We implement XSCOPE in Python, C++, and CUDA. We use the Bayesian Optimization Python package [14] to implement the BO search, which depends on Numpy, Scipy, and Scikit-learn. We compile the C++ examples with the Intel compiler, and the CUDA examples with the NVIDIA nvcc compiler, using `clang/LLVM` as the host compiler. While we focus on CUDA and NVIDIA GPUs, XSCOPE can be applied to other classes of GPUs, such as AMD and Intel GPUs.

**Acquisition Functions.** We evaluate the approach under three acquisition functions: Upper Confidence Bound (UCB), Expected Improvement (EI), and Probability of Improvement (PI) [6,10], which are widely used in BO. We set the acquisition functions to use "exploration" mode—since input bounds are very large (essentially unbounded), "exploration" increases the chances of identifying extreme cases (compared to the "exploitation" mode).

### 3.7. Limitations

Our approach is not without limitations. The most important limitations are the following:

- While our approach is designed to systematically search for infinity and subnormal numbers, it not specifically designed to trigger `NaN`. However, as Section 4 shows, XSCOPE has proven to be useful in identifying inputs that trigger `NaN` in many of the evaluated functions. The main reason for this is that several `NaNs` are the result of calculations that involve infinity; for example the

following expressions generate NaN: $0 \times \texttt{INF}$, $\frac{0}{\texttt{INF}}$, and $\texttt{INF} - \texttt{INF}$. Therefore, while NaNs are not directly targeted by BO, it can identify inputs that generate them and reports them to the user.

- The run time of the *many-range* approach can significantly increase with the number of input parameters because it tries many combinations of small input bounds for different combinations of parameters. The approaches with fewer bounds to try (*two-range* and *whole-range*) are much faster but may not find as many exceptions as the *many-range* approach. We experimented with functions with 1–3 inputs—we found that for functions with 4 or more inputs, the *many-range* approach would take more than 12 h (the maximum allocation time of our clusters). In future, work we plan to study methods to improve the scalability of *many-range*.
- Because we assume that functions to test are black boxes, Xscope cannot identify exceptions that occur internally in the function but do not propagate to the return value. While we believe this is rare in most functions, it could happen, for example, when a value v is computed and used only in a branch (e.g., `if(v) {...}`), but v does not affect directly the final computed value. We are working on developing a binary-analysis method (that uses NVBit [15]) which is able to detect exceptions at the binary inside the tested function while BO runs. We leave studying this improvement in Xscope to future work.

## 4. Evaluation

Our evaluation aims to answer the following questions:

**Q1:** Can Xscope identify inputs that trigger exceptions in black-box CPU or GPU functions?

**Q2:** What approach for sampling and input splitting works best to search for such inputs?

**Q3:** Can Xscope identify exception-triggering inputs that impact a CPU function but not the GPU counterpart, or vice versa?

**Q4:** How does the performance of Xscope compares to random sampling?

### 4.1. Tested functions and programs

**CUDA Math Library.** NVIDIA provides several proprietary accelerated libraries, such as cuBLAS, cuFFT, cuSOLVER, CUDA Math Library, and cuDNN, that are widely used in closed- and open-source software. We evaluate Xscope on the double-precision functions provided in the CUDA Math Library[8]; these functions are widely used in scientific software, the source code is not publicly available, and the documentation about expected inputs is limited (the NVIDIA documentation, however, informs the user of expected return values for **some** inputs).

While all functions return a double-precision value, we analyze only the functions that receive scalar double-precision parameters—we omit functions that use integer parameters or arrays as input. In total, we analyze 58 functions from the CUDA Math Library (about 68% of the library functions).

**Intel Math Library.** We evaluate Xscope in the Intel C++ Compiler Classic Math Library.[9] In contrast to the CUDA Math Library functions that run in NVIDIA GPUs, the Intel Math Library function run a CPU. Several of the functions provided in the CUDA Math Library are also provided in the Intel library. This allows Xscope check if similar functions behave equally on different platforms (despite being developed by different vendors). In total, we analyze 81 functions from the Intel Math Library.

[8] https://developer.nvidia.com/cuda-math-library

[9] https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler-documentation.html

**Table 3**
Comparison of coverage metrics for three acquisition functions: Upper Confidence Bound (UCB), Expected Improvement (EI), Probability of Improvement (PI).

| Acquisition function | Total exceptions | Exception-triggering functions | Exception types |
|---|---|---|---|
| UCB | 1303 | 44 | 5 |
| EI | 1305 | 44 | 5 |
| PI | 1305 | 44 | 5 |

### 4.2. System

**NVIDIA GPU Experiments.** For the GPU experiments in CUDA, we use an IBM Power9 system with 44 ppc64le Cores/Node, 4 NVIDIA V100 GPUs per node, 256 GB of CPU Memory/Node, with 795 nodes in total. The system uses Linux 4.14.0-115.21.2.1chaos.ch6a.ppc64le, CUDA version 10.1.243 and Python 3.7.2 compiled with GCC 4.9.3.

**Intel CPU Experiments.** For the Intel CPU experiments, we an Intel Xeon E5-2695 v4, with 36 cores/node, 128 GB of CPU Memory/Node, with 3,018 nodes in total. The system uses Linux 3.10.0-1160.76.1.1chaos.ch6.x86_64, and Python 3.7.2 compiled with GCC 4.9.3, and the Intel compiler version 19.0.4.227.

### 4.3. CUDA math library

We start by analyzing the impact of using different maximum iteration bounds for BO and the acquisition functions used in BO, since this can impact the rest of the experiments. We start with the CUDA Math Library since errors in the GPU have a higher impact (NVIDIA GPUs have no mechanisms to detect exceptions).

**Iterations Analysis.** Fig. 6 shows a comparison of the approaches for sampling and input splitting for different maximum iterations settings for BO, using the EI acquisition function. Fig. 6(a) shows the total number of inputs found; since the methods have different sampling numbers, we normalize each method by dividing the total exception-triggering inputs found by the maximum number of samples allowed. Fig. 6(b) shows the number of different exception-triggering functions from the CUDA Math library.

From Fig. 6 we observe the following. First, the number of maximum BO iterations do not affect the results significantly when we vary this maximum between 5–30. Second, the *whole-range* method appears to be effective in identifying exception-triggering inputs per trial (or sample) when one looks at the total counts. While *whole-range* can find many inputs that trigger a single exception class (e.g., INF), it can get stuck easily. On the other hand, the *many-range* approach is more expensive, but does not get stuck easily and can find many more different classes of exception-triggering inputs on different functions.

**Acquisition Function Results.**
Table 3 compares three metrics for three different acquisition functions: UCB, EI, and PI [6,10] using 30 max samples for BO and the *many-range* method. As described in Section 3, the acquisition functions use the *exploration* mode. We observe no significant difference in the results: EI and PI are slightly better than UCB in terms of total exceptions found, but they all identify the same number of exception-triggering functions and exception types. The rest of the experiments focus on using the EI function.

**Coverage Results.** We now dig into coverage results for the individual functions of the CUDA Math Library. Fig. 7 presents the number of exception-triggering inputs for the two approaches to sample numbers: *fp* and *exp* method. We only present the results for the *whole-range* and *many-range* approach. We evaluated the *two-range* approach and the results sit in between the whole-range and many-range, so we left them out to save space.

We observe that, while our BO search does not target NaNs, it is capable to find inputs that generate NaNs in many functions. Xscope is

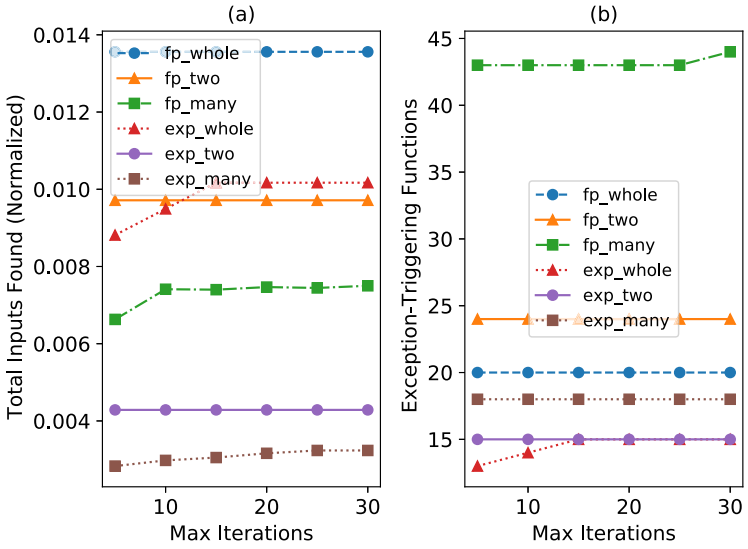


**Fig. 6.** (a) Total exception-triggering inputs divided by the maximum number of samples allowed; (b) Number of different exception-triggering functions from the CUDA Math Lib.

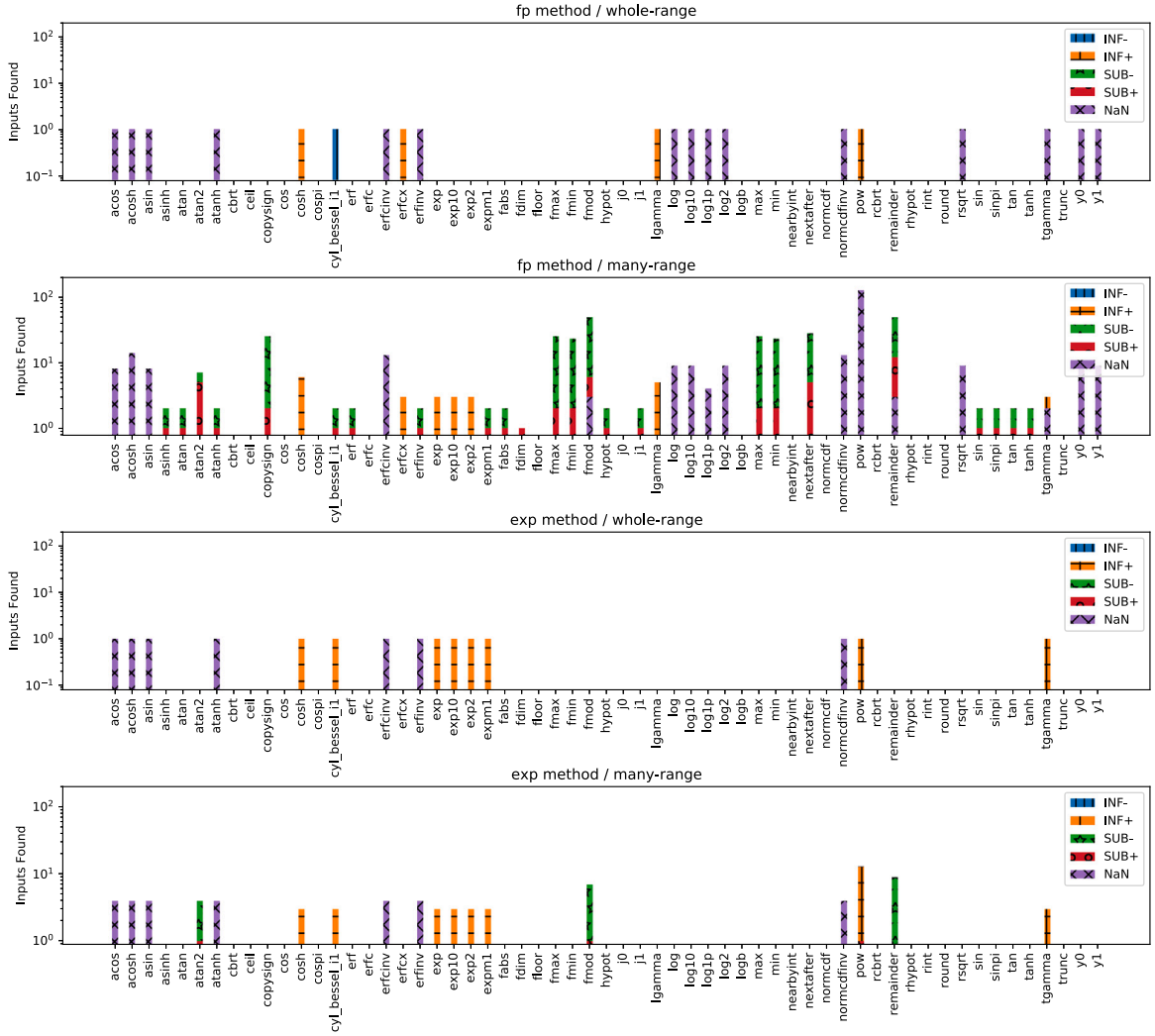


**Fig. 7.** Number of inputs found for the CUDA Math Library functions. SUB+ mans positive subnormal and SUB- means negative subnormal.

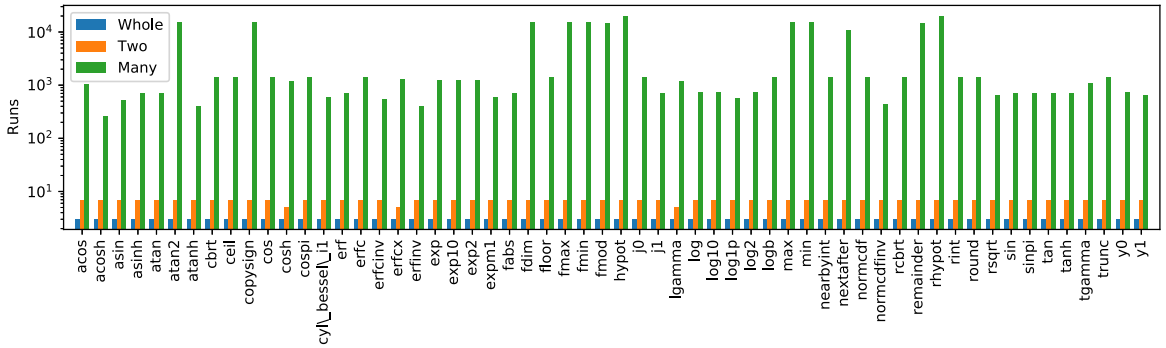


**Fig. 8.** Number of runs of the target $f$ for the *fp+many-range* approach.

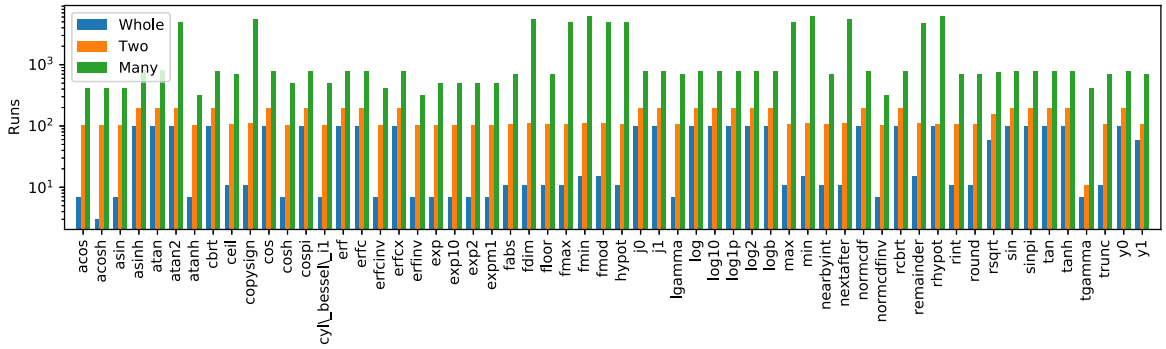


**Fig. 9.** Number of runs of the target $f$ for the *exp+many-range* approach.

able to find inputs that generate `NaN` in 15 functions from the library (25% of the tested library functions). We also observe that the many-range approach to split input ranges is capable of finding offending inputs in more cases than the whole-range approach.

In general, we find that the *fp* method for sampling numbers is able to identify more offending inputs than the *exp* method. There are a few cases, however, where *exp* finds offending inputs but *fp* cannot. For example, *exp+many-range* finds inputs that generate `NaN` in the `erfinv` function but the *fp+many-range* cannot.

**Outcome 1:** `XSCOPE` proves to be effective at identifying inputs that trigger exceptions in black-box CUDA functions with limited information of input bounds. It is able to finds triggering inputs in 72% of the tested functions in the CUDA Math Library.

**Performance Results.** The different approaches we try trade off speed versus coverage. To understand this trade-off, we show in Fig. 9 and Fig. 8 the number of runs (of the target function) that BO executes, which is the factor that dominates execution time. As we expected, the many-range method finds more offending inputs than the other methods at the expense of many more runs of the target function. In light of these results, we expect that users will consider the trade-off of coverage versus runtime when using `XSCOPE`.

**Outcome 2:** The *many-range* approach to split input bounds identifies inputs that trigger exceptions many more times than the *whole-range* and *two-range* approach; however, this comes at the price of significantly more execution runs of the target. In general, the *fp* approach to sample numbers works better in practice than the *exp* approach; however, *exp* is successful a few times where *fp* is not.

**Execution Time.**
Fig. 10 shows the execution time of `XSCOPE` in finding inputs for the *fp* method (we omit the *exp* approach here as it provides inferior results). The plots are sorted by the values of the *Many* method. The execution time for a given function is dominated by two factors: (1) the function execution time, and (2) the time to update the BO optimizer, and (3) the number of times the optimizer runs. The function execution time varies depending on its code—a black box from the point of view of `XSCOPE`. We have measured the time to update the BO optimizer, and have found that it is mostly constant, in the order of 0.027 s for the BO library we use (this time may change with a different library). Finally, the number of times the optimizer runs is presented in Figs. 8. The larger execution times (order of minutes) correspond to functions with two parameters (e.g., `hypot`).

### 4.4. Example: `cosh(double x)`

As an example of specific inputs found by `XSCOPE`, consider the `cosh(double x)` from the CUDA Math Library, which calculates the hyperbolic cosine of the input argument $x$. This is an increasing function, which is expected to produce `INF` as input values increase. However, from the library documentation it is not very clear what specific inputs would produce `INF`. Particularly, the library documentation specifies that the function returns:

- 1.0 for `cosh(0)`
- `INF+` for `cosh(INF±)`

`XSCOPE` finds specific inputs that trigger `INF+`, such as 4.35e+3, 1e+47, and 4.17+306. This provides to users more specific information than that provided in the documentation. We observe similar benefits in other functions, such as `erfinv()`, where `XSCOPE` improves the user's knowledge over what is provided in the documentation regarding exception-inducing inputs.

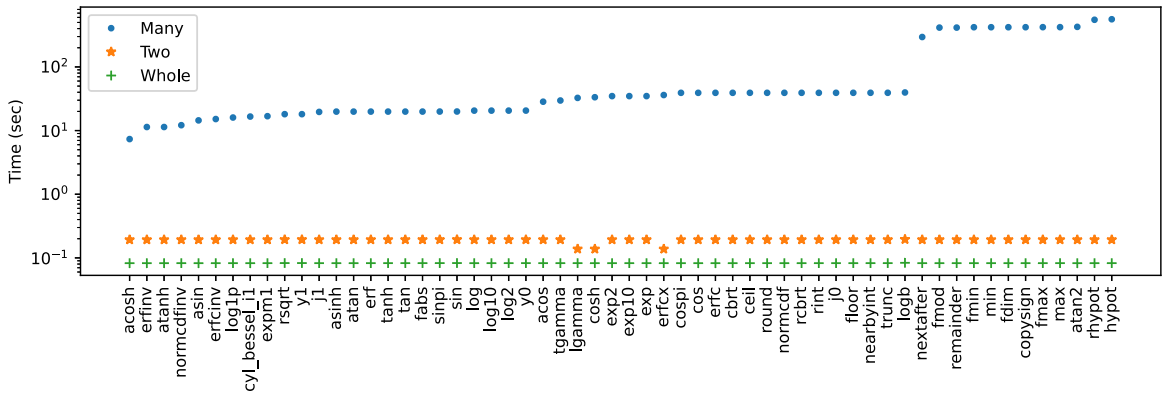


**Fig. 10.** Execution time (seconds) of the *fp* approach for sampling. The plots are sorted by the values of the *Many* method. The larger execution times (order of minutes) correspond to functions with two parameters (e.g., `hypot`).

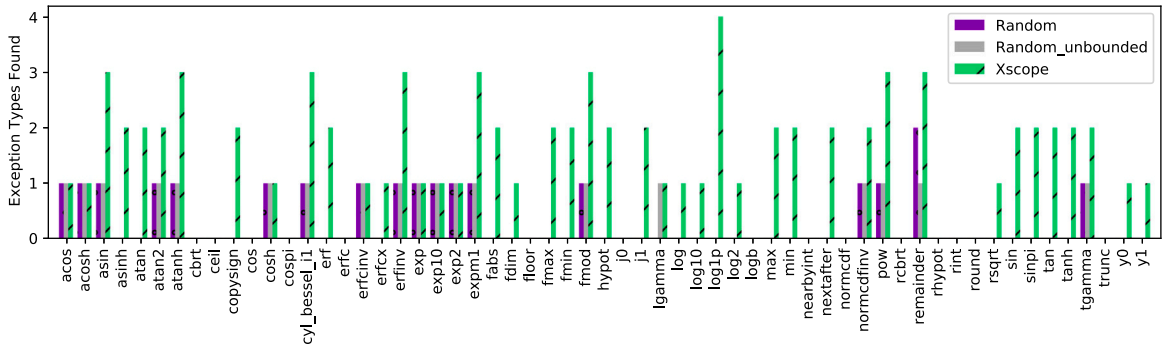


**Fig. 11.** Comparison of `Xscope` with two random sampling approaches: Random stops the input search when the first exception is found (this is how `Xscope` operates); Random_unbounded does not stop when an exception is found.

### 4.5. Comparison to random sampling

We compare `Xscope` (with *fp* and *whole-range*) to *random sampling*, which would be the only method available to programmers to find exception-triggering inputs in black-box functions. We use two approaches: (1) *Random*, which stops sampling inputs when the first exception is found—note that this is how `Xscope` operates as well; (2) *Random_unbounded*, which does not stop sampling when an exception is found. For each method, we set the maximum number of samples to be the same, so the all have equal chances to find offending inputs. Fig. 11 shows the number of different exceptions types found (the maximum is five) for each function.

**Outcome 3:** We observe that `Xscope` is superior than random sampling as it is able to identify many more different exception types in most functions than random sampling.

### 4.6. Intel math library

Fig. 12 shows the number of inputs found for the Intel Math Library functions for the five categories of exceptions. `Xscope` identifies inputs that trigger exceptions in 62 functions out of 81 tested functions (i.e., about 75% of the tested functions). The effectiveness of `Xscope` in the CPU functions is comparable of that for GPU functions, where exception-triggering inputs were found in 72% of the CUDA functions.

The Intel and CUDA libraries provide several similar functions. Scientific applications can use a given math function, e.g., `sqrt()`, in the code with the expectation that the compiler and linker defines the actual implementation used at runtime, depending on where the function is executed (CPU or GPU).

Table 4 shows a comparison of the inputs found for the same function executed in the CPU or GPU. The CPU version is compiled with the Intel compiler and the GPU version is compiled with the NVIDIA compiler. As expected, most functions exhibit similar behavior, i.e., `Xscope` identifies the same number of inputs that trigger exceptions. In some cases, however, we observe some differences. For example, for functions `y0` and `y1`, `Xscope` finds inputs that produce `INF`- in the CPU, not in the GPU, while the same functions produce `NaN` in the GPU, but not in the CPU.

**Outcome 4:** While most functions executed in CPUs and GPUs behave in similar ways with respect to inputs that trigger exceptions, `Xscope` identified a few cases where the behavior is inconsistent between the CPU and GPU implementation.

Such findings are important in order to alert designers who may port code from CPUs to GPUs: they must plan for such exception differences in the overall code.

## 5. Related work

**Floating-Point Error-Inducing Input Generation.** Several previous methods have been proposed to generate inputs that induce floating-point error. S3FP [16] is a tool for determining the input settings to a floating point routine that maximizes its result error,

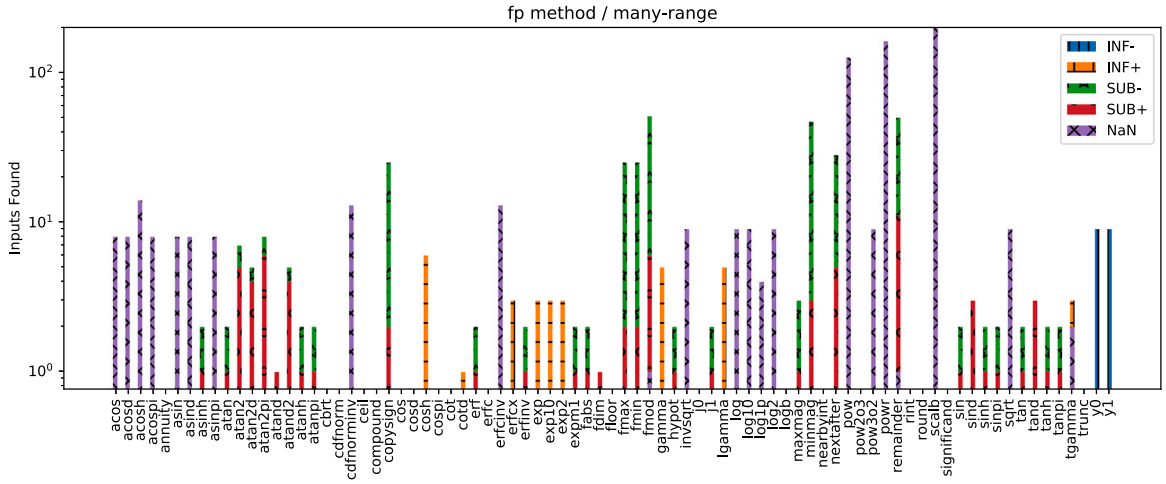


**Fig. 12.** Number of inputs found for the Intel Math Library functions for the five categories of exceptions.

using a heuristic search algorithm called Binary Guided Random Testing (BGRT). S3FP iteratively divides the search range of each input variable into two and permutes them randomly to generate a tighter search space. The tool evaluates each subspace by sampling inputs and selecting one for further exploration. FPGen [17] formulates the problem of generating high error-inducing floating-point inputs as a code coverage maximization problem and solves it using symbolic execution, focusing on detecting large precision loss and cancellation. FPGen applies symbolic execution to generate inputs that exercise specialized branches that when covered by a given input, are likely to lead to large errors in the result. In [18] authors propose a heuristic search-based approach to automatically generating test inputs that aim to trigger significant inaccuracies in floating-point programs. It builds a reliable fitness function to guide the search. The work in [19] proposes a testing approach to trigger high floating-point inaccuracies by utilizing heuristic rules drawn from error analysis to guide the process of global search of test cases. It compares the approach to random and BGRT-based methods and show its stability. The methods in this domain attempt to maximize or induce floating-point rounding error and inaccuracies, such as cancellation; however, they do not target triggering exceptions, are not available in GPU programs and require analyzing the source code.

The closest work to Xscope is perhaps Ariadne [3], which transforms a program to explicitly check exception triggering conditions, by symbolically executing the transformed program using real arithmetic to find candidate real-valued inputs that can reach and trigger an exception. The main difference with respect to our work is this approach is designed to operate in the CPU (host) code and requires analyzing the source code via symbolic execution.

**Floating-Point Exception Detection.** A number of tools have been designed to detect floating-point exceptions at runtime. The goal of these tools is not to find inputs that trigger exceptions but to detect them when the program is run with a specific input. FPSpy [1] monitors floating-point behaviors in CPUs using operating-system facilities and hardware flags; it reports floating-point exceptions at the binary level for x86 binaries. FPChecker [2] performs exception detection in CUDA programs at the LLVM level and reports the root cause of the exception (file and line numbers) at runtime, while GPU-FPX [20] is a similar tool that works based on binary instrumentation of NVIDIA SASS instructions. Other tools such as CADNA [21] and Verificarlo [22] report large cancellations in an addition or subtraction—but not other forms of exceptions. The work in [23] examines the ways in which different exceptions are handled in numeric programs (particularly overflow and underflow), with emphasis on making numeric code run faster. Exception scenarios are handled purely in numerical programs because often programmers do not understand floating-point arithmetic—researches in [24] conduct a study of different groups from academia, national labs, and industry, and found that developers poorly understood which compiler optimizations were non-standard.

## 6. Conclusion

In this work, we demonstrate that Bayesian optimization (BO) can be applied systematically to find inputs that trigger floating-point exceptions in black-box functions, where the source is unavailable and users have limited information about the expected function inputs. We have implemented various methods for number sampling and input exploration to guide BO in triggering such exceptions. To the best of our knowledge, we are the first to apply BO and such methods to identify exception-triggering inputs. We present an algorithm and the implementation of ideas in Xscope. While we evaluate Xscope in NVIDIA GPUs and Intel CPUs, the situation in CPU programs is less complicated than in GPUs, because CPU architectures (e.g., x86) traditionally provide hardware register flags for exceptions, whereas NVIDIA GPUs and CUDA do not offer support for exception detection. We demonstrate Xscope in 58 functions from the CUDA Math Library and 81 functions from the Intel Math Library. Xscope is able to identify inputs that trigger exceptions in about 72% of the CUDA tested functions, and in about 75% of the Intel Math Library. When Xscope is compared to random sampling, Xscope triggers a significantly larger number of exceptions, proving its superiority to random sampling.

## Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Ignacio Laguna reports financial support was provided by US Department of Energy. Ignacio Laguna reports a relationship with US Department of Energy that includes: funding grants.

## Data availability

Data will be made available on request.

## Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments and helpful suggestions. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory, United States under Contract DE-AC52-07NA27344 (LLNL-JRNL-852650), DOE ASCR Award Number DE-SC0022252 and NSF CISE Awards 1956106, 2124100, and 2217154.

**Table 4**
Comparison of inputs that trigger exceptions in CPU (Intel Math Library) and GPU (CUDA Math Library) found by XSCOPE.

| Func. | CPU | | | | | GPU | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | INF- | INF+ | SUB- | SUB+ | NaN | INF- | INF+ | SUB- | SUB+ | NaN |
| acos | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 8 |
| asin | 0 | 0 | 2 | 1 | 8 | 0 | 0 | 2 | 1 | 8 |
| atan | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 2 | 1 | 0 |
| atan2 | 0 | 0 | 7 | 5 | 0 | 0 | 0 | 7 | 5 | 0 |
| cos | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| cospi | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| sin | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 2 | 1 | 0 |
| sinpi | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 2 | 1 | 0 |
| tan | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 2 | 1 | 0 |
| acosh | 0 | 0 | 0 | 0 | 14 | 0 | 0 | 0 | 0 | 14 |
| asinh | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 2 | 1 | 0 |
| atanh | 0 | 1 | 2 | 1 | 0 | 0 | 1 | 2 | 1 | 0 |
| cosh | 0 | 6 | 0 | 0 | 0 | 0 | 6 | 0 | 0 | 0 |
| tanh | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 2 | 1 | 0 |
| cbrt | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| exp | 0 | 3 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 |
| exp10 | 0 | 3 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 |
| exp2 | 0 | 3 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 |
| expm1 | 0 | 1 | 2 | 1 | 0 | 0 | 1 | 2 | 1 | 0 |
| hypot | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 2 | 1 | 0 |
| log | 0 | 0 | 0 | 0 | 9 | 0 | 0 | 0 | 0 | 9 |
| log10 | 0 | 0 | 0 | 0 | 9 | 0 | 0 | 0 | 0 | 9 |
| log1p | 1 | 0 | 2 | 1 | 4 | 1 | 0 | 2 | 1 | 4 |
| log2 | 0 | 0 | 0 | 0 | 9 | 0 | 0 | 0 | 0 | 9 |
| logb | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| pow | 0 | 50 | 0 | 2 | 126 | 0 | 50 | 0 | 1 | 126 |
| erf | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 2 | 1 | 0 |
| erfc | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| erfcx | 0 | 3 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 |
| erfcinv | 0 | 0 | 0 | 0 | 13 | 0 | 0 | 0 | 0 | 13 |
| erfinv | 0 | 1 | 2 | 1 | 0 | 0 | 1 | 2 | 1 | 0 |
| j0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| j1 | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 2 | 1 | 0 |
| lgamma | 0 | 5 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 |
| tgamma | 0 | 3 | 0 | 0 | 2 | 0 | 3 | 0 | 0 | 2 |
| y0 | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 9 |
| y1 | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 9 |
| ceil | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| floor | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| nearbyint | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| rint | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| round | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| trunc | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fmod | 0 | 0 | 51 | 6 | 1 | 0 | 0 | 49 | 6 | 3 |
| remainder | 0 | 0 | 50 | 11 | 1 | 0 | 0 | 49 | 12 | 3 |
| copysign | 0 | 0 | 25 | 2 | 0 | 0 | 0 | 25 | 2 | 0 |
| fabs | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 2 | 1 | 0 |
| fdim | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| fmax | 0 | 0 | 25 | 2 | 0 | 0 | 0 | 25 | 2 | 0 |
| fmin | 0 | 0 | 25 | 2 | 0 | 0 | 0 | 23 | 2 | 0 |
| nextafter | 0 | 0 | 28 | 5 | 0 | 0 | 0 | 28 | 5 | 0 |

## References


[1] P. Dinda, A. Bernat, C. Hetland, Spying on the floating point behavior of existing, unmodified scientific applications, in: Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing, 2020, pp. 5–16.

[2] I. Laguna, Fpchecker: Detecting floating-point exceptions in GPU applications, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE, 2019, pp. 1126–1129.

[3] E.T. Barr, T. Vo, V. Le, Z. Su, Automatic detection of floating-point exceptions, ACM Sigplan Not. 48 (1) (2013) 549–560.

[4] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, E. Shelhamer, CUDNN: Efficient primitives for deep learning, 2014, arXiv preprint arXiv:1410.0759.

[5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al., Pytorch: An imperative style, high-performance deep learning library, in: Advances in Neural Information Processing Systems, Vol. 32, 2019.

[6] J. Snoek, H. Larochelle, R.P. Adams, Practical bayesian optimization of machine learning algorithms, in: Advances in Neural Information Processing Systems, Vol. 25, 2012.

[7] H. Ha, S. Rana, S. Gupta, T. Nguyen, S. Venkatesh, et al., Bayesian optimization with unknown search space, Adv. Neural Inf. Process. Syst. 32 (2019).

[8] B. Shahriari, A. Bouchard-Côté, N. Freitas, Unbounded Bayesian optimization via regularization, in: Artificial Intelligence and Statistics, PMLR, 2016, pp. 1168–1176.

[9] I. Laguna, G. Gopalakrishnan, Finding inputs that trigger floating-point exceptions in GPUs via Bayesian optimization, in: 2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, SC, IEEE Computer Society, 2022, pp. 454–467.

[10] F. Archetti, A. Candelieri, Bayesian Optimization and Data Science, Springer, 2019.

[11] H. Guo, I. Laguna, C. Rubio-González, PLINER: Isolating lines of floating-point code for compiler-induced variability, in: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2020, pp. 1–14.

[12] V. D'Silva, L. Haller, D. Kroening, M. Tautschnig, Numeric bounds analysis with conflict-driven learning, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2012, pp. 48–63.

[13] G. Guennebaud, B. Jacob, et al., Eigen v3, 2010, http://eigen.tuxfamily.org.

[14] F. Nogueira, Bayesian Optimization: Open source constrained global optimization tool for Python, 2014, URL https://github.com/fmfn/BayesianOptimization.

[15] O. Villa, M. Stephenson, D. Nellans, S.W. Keckler, NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs, in: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019, pp. 372–383.

[16] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, A. Solovyev, Efficient search for inputs causing high floating-point errors, in: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, 2014, pp. 43–52, http://dx.doi.org/10.1145/2555243.2555265.

[17] H. Guo, C. Rubio-González, Efficient generation of error-inducing floating-point inputs via symbolic execution, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20, 2020, pp. 1261–1272, http://dx.doi.org/10.1145/3377811.3380359.

[18] D. Zou, R. Wang, Y. Xiong, L. Zhang, Z. Su, H. Mei, A genetic algorithm for detecting significant floating-point inaccuracies, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 1, IEEE, 2015, pp. 529–539.

[19] X. Yi, L. Chen, X. Mao, T. Ji, Efficient global search for inputs triggering high floating-point inaccuracies, in: 2017 24th Asia-Pacific Software Engineering Conference, APSEC, IEEE, 2017, pp. 11–20.

[20] X. Li, I. Laguna, B. Fang, K. Swirydowicz, A. Li, G. Gopalakrishnan, Design and evaluation of GPU-FPX: A low-overhead tool for floating-point exception detection in NVIDIA GPUs, in: Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing, in: HPDC '23, Association for Computing Machinery, New York, NY, USA, ISBN: 9798400701559, 2023, pp. 59–71, http://dx.doi.org/10.1145/3588195.3592991.

[21] F. Jézéquel, J.-M. Chesneaux, CADNA: A library for estimating round-off error propagation, Comput. Phys. Comm. 178 (12) (2008) 933–955.

[22] C. Denis, P.D.O. Castro, E. Petit, Verificarlo: Checking floating point accuracy through Monte Carlo arithmetic, 2015, arXiv preprint arXiv:1509.01347.

[23] J.R. Hauser, Handling floating-point exceptions in numeric programs, ACM Trans. Program. Lang. Syst. (TOPLAS) 18 (2) (1996) 139–174.

[24] P. Dinda, C. Hetland, Do developers understand IEEE floating point? in: 2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS, IEEE, 2018, pp. 589–598.